CS2141 – Software Development using C/C++

C++ Basics

Integers – Basic Types

- Can be **short**, **long**, or just plain **int**
- C++ does not define the size of them other than **short <= int <= long**
  - They could all be the same size
  - Commonly at least two of them are the same size
  - The **sizeof** operator can be used to find out the size:

```
cout << "A short int is " <<
        sizeof( short int ) <<
        " bytes" << endl;
```

Integers – Signed and Unsigned

- Integers can also be `signed` or `unsigned`
- Unsigned integers use the sign bit for the number
  - An `unsigned int` can only hold positive numbers
  - An `unsigned int` will hold a bigger positive number than a `signed int`
- Integers are signed by default
- An `unsigned long int` holds the largest positive integer value
- A `signed short int` is the "shortest"

Integers – Division and Modulus

- C++ leaves the outcome of a few operations up to the platform
- Integer division and modulus with negative operands are two of those unspecified operations
- **`-23 / 4`** could be **`-5`** or **`-6`**
- **`-23 % 4`** could be **`-3`** or **`1`**
- It will always be true that:
  `a == (a / b) * b + a % b`

Characters – Basic Types

- A **`char`** is typically only 8 bits
  - C++ only defines a minimum length, so longer characters are allowed
  - A **`w_char`** is longer than a **`char`**, usually the same as a **`short`**
- Characters can be **`signed`** or **`unsigned`**
  - **`char`** is unsigned by default
  - **`signed char`** can be used to store small integers

Using Characters

- Characters can be used in arithmetic expressions:

```
char c = ' ' + '!';  // c will be 'A'
int x = '9' - '0';   // x will be 9
```

- There are many ways to represent a character:
  - A character: `'n'`
  - ASCII: `'\156'`
  - Hexadecimal (note the 0x prefix): `'\0x6e'`
  - An integer: `110`
- Strings are often stored as an array of characters, terminated by a `'\0'` (the null character).

Booleans

- A `bool` is a single bit:
  - 1 for true
  - 0 for false
- A `bool` can be used as an integer:

```
bool test = true;
int i = 2 + test;
test = test - 1;
```

- The `bool` type is relatively new to C++. There used to be various competing designs, which might be encountered in older code.

Using Integers as Booleans

- Integers are often used as a boolean type:
  - Zero is false
  - Any other value is true

```
int i = 10;
while( i )
{
    // Do something
    // until i is 0
    i--;
}
```

Real Numbers – Basic Types

- Can be **`float`**, **`double`**, or **`long double`**
  - **`float`** is the smallest
  - **`long double`** is the biggest
- Most math libraries use doubles, so it is better to use **`double`** rather than **`float`**
- Some platforms may provide values like **`Nan`**, **`NEGATIVE_INFINITY`**, and **`POSITIVE_INFINITY`**, but they are not required by the language

Conversion Between Data Types

- C++ will convert operand data types if necessary:

```
int i;
double d = 3.14159;
i = d;          // May create a warning,
                // but it will work.
i = (int)d;     // Use cast to avoid warning.
```

- Be aware of data types in expressions:

```
int a = 3;
int b = 2;
float c = (a + b) / 2; // 2.0, not 2.5
```

Enumerations

- An *enumeration* creates a distinct integer type with named values:

```
enum color { red, orange, yellow };
color bgColor = red;
if( bgColor == orange ) ...
```

- Each of the names can only be used once in any specific namespace:

```
// This will cause an error
enum fruit { apple, pear, orange };
```

- Integer values can be specified. If a value is not provided, the previous value is incremented:

```
enum axes { X = 0, Y = 1, Z = 2 };
enum letters { A = 0, B, C };
```

Basic Stream I/O

```
#include <iostream> // I/O function definitions
using namespace std;

int a, b;          // Variable declarations

// Basic integer input
cin >> a >> b;

// Basic string output
cout << "Hello world" << endl;
cout << a << " + " << b << " = " <<
a + b << endl;
```

Declaring Arrays

- Arrays are declared by the name and the number of elements
- The **new** directive does not have to be used to allocate an array
- The number of elements can be omitted if there is a way for the compiler to determine it

```
int data[100];
char text[] = “This is an example of an array”;
int evens[] = {2, 4, 6, 8, 10, 12, 16, 18};
```

- C++ arrays do not know their own size, you must keep track of it yourself

Using Arrays

- The number of elements can also be omitted if the array is passed as an argument to a function:

```
double average( int n, double data[] )
{
   double sum = 0;
   for( int i = 0; i < n; i++ )
   {
      sum += data[i];
   }
   return sum / n;
}
```

- Notice that the parameter **n** is used to pass the size of the array

Working with Objects

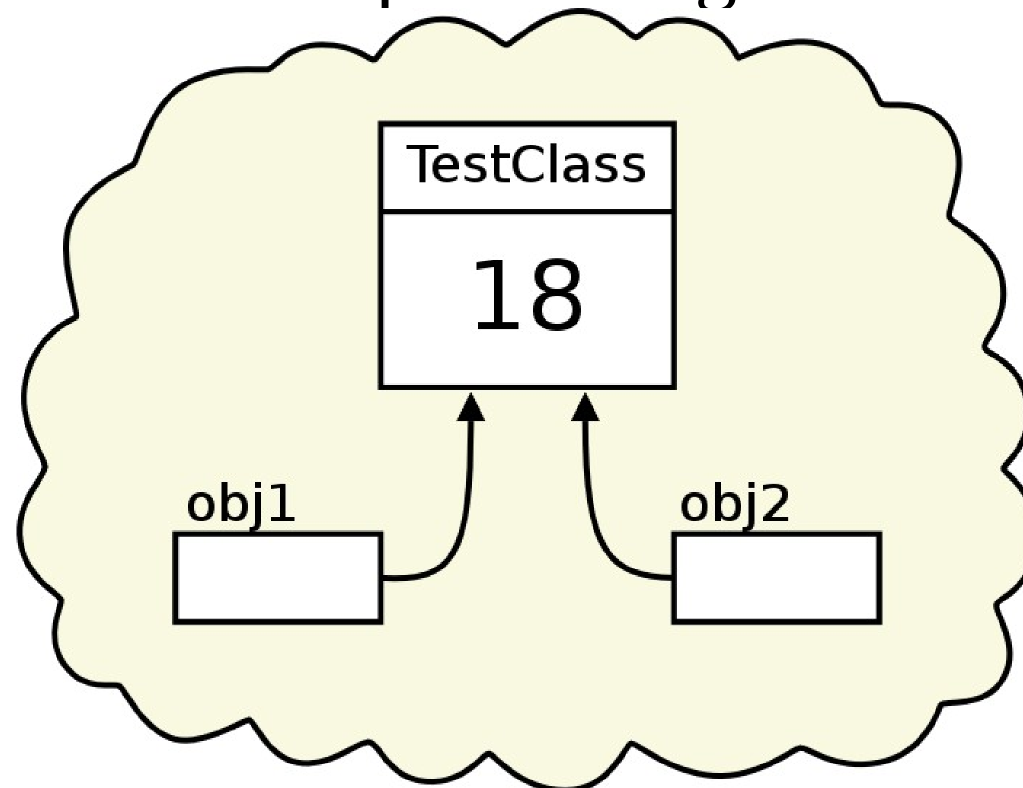
- Consider the following Java code:

```
public class TestClass
{
   public int value;

   public static void main( String[] args )
   {
     TestClass obj1 = new TestClass( );
     TestClass obj2;
     obj1.value = 12;
     obj2 = obj1;
     obj1.value = 18;
     System.out.println( "obj1 value " + obj1.value );
     System.out.println( "obj2 value " + obj2.value );
   }
}
```

Working with Objects cont.

- Java uses *reference sematics* for assignments, so when the code is run, both **`obj1`** and **`obj2`** are variables that end up referring to the same object:



Working with Objects cont.

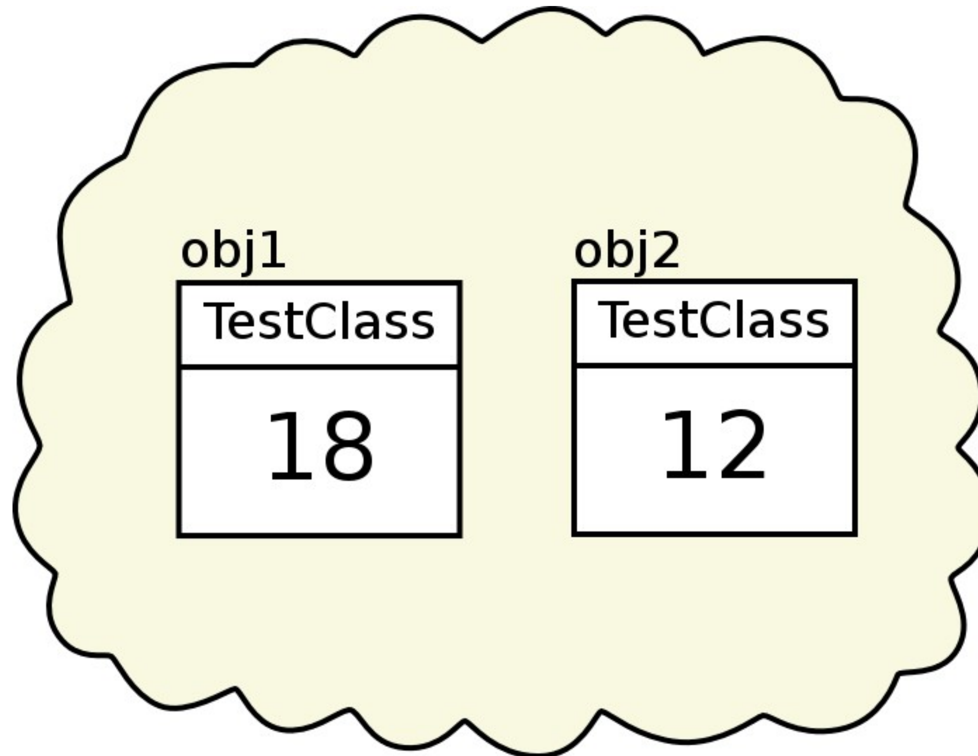
- Now consider the C++ version:

```
class TestClass
{
   public:
     int value;
};

int main( )
{
   TestClass obj1;
   TestClass obj2;
   obj1.value = 12;
   obj2 = obj1;
   obj1.value = 18;
   cout << "obj1 value " << obj1.value << endl;
   cout << "obj2 value " << obj2.value << endl;
}
```

Working with Objects cont.

- C++ uses *copy semantics* for assignments, so when the code is run **`obj1`** and **`obj2`** are two different objects with different values:



Working with Objects cont.

- Difference:
  - In the Java version, **`obj1`** and **`obj2`** are references to a TestClass object
  - In the C++ version, **`obj1`** and **`obj2`** are TestClass objects
- If access by reference is needed, it is left to the programmer in C++ (more on references later):

```
TestClass obj1;
obj1.value = 12;
TestClass & obj2 = obj1;
obj1.value = 18;
```

Function Definitions

- C++ allows functions to be defined outside of classes. These are called *global functions*
- Functions are invoked by using their name.

```
int max( int i, int j )
{
    if( i < j )
      return j;
    return i;
}
int x = 283;
int y = 482;
int z = max( x, y );
```

Function prototypes

- A *function prototype* simply defines the name and argument types of a function
  - There is no function body
  - Argument names can be used but are not required
- Prototypes are necessary because the compiler must know a function exists before the function can be invoked

- The prototype for the **max** function would be:

```
int max( int, int );
```

The `main` function

- Execution of a C++ program begins in the function **`main`**
  - **`main`** is **not** part of any class
  - It should **not** be declared static
- The return type must be **`int`**
  - Older compilers might accept **`void`**
  - Returning zero means successful completion
  - The meaning of other values is up to the compiler or even the programmer

The `main` function cont.

- There can be zero or two parameters:
  - Zero parameters:
    ```
    int main( )
    ```
  - Two parameters:
    ```
    int main( int argc, char ** argv )
    ```
    - The first parameter is an integer passing the number of arguments to the program.
    - The second parameter is an array of strings passing any command-line arguments to the program.
- The first argument to a C++ program (`argv[0]`) is always the program name.

“Hello World” - revisited

```
#include <iostream>
using namespace std;

int main( int argc, char ** argv )
{
   cout << “Hello World!” << endl;
   cout << “From “ << argv[0] << endl;
   return 0;
}
```